Compladni: Efficient Computational Simulation of Chladni Patterns from Raw Audio

1 Introduction

The visualization of sound through physical patterns has long captivated scientists and artists alike. Chladni patterns, first documented by Ernst Chladni in the 18th century, represent one of the most fascinating demonstrations of wave mechanics in physical systems [1]. These patterns emerge when a plate covered with fine particles vibrates at specific frequencies, causing the particles to accumulate along nodal lines where no vibration occurs, thereby creating distinctive geometric patterns.

The theoretical foundation for understanding these patterns was established through extensive research in plate vibrations and wave mechanics. Colwell et al. demonstrated the mathematical principles governing the formation of Chladni figures on square plates [2], and later expanded this work to include comprehensive analyses of vibrations in symmetrical plates and membranes [3]. Their work provides the fundamental equations that describe how standing waves create these intricate patterns.

Recent technological advances have enabled new approaches to studying and visualizing these acoustic phenomena.[4] [5] While traditional Chladni patterns are created through direct mechanical vibration of plates, modern computational methods offer the possibility of simulating these patterns from any audio source, particularly music. This intersection of classical physics, digital signal processing, and computational visualization presents an opportunity to create novel representations of musical compositions.

Our research focuses on developing a computational framework for transforming musical pieces into dynamic Chladni patterns through simulation. This approach combines audio signal processing techniques, physical modeling of wave mechanics, and advanced visualization methods. By analyzing the frequency components and temporal dynamics of music using Fast Fourier Transform (FFT) and implementing physics-based particle simulation, we aim to create physically accurate and visually engaging representations of sound.

The significance of this work extends beyond mere visualization. It bridges the gap between acoustic physics and digital art, providing new insights into the relationship between musical structure and physical wave phenomena. Furthermore, this research contributes to the broader field of scientific visualization by developing methods for representing complex acoustic data in an intuitive and aesthetically pleasing manner.

This paper presents a comprehensive approach to simulating Chladni patterns from musical input, encompassing audio analysis, physics simulation, and visualization techniques. We build upon established research in plate vibrations and wave mechanics while introducing novel computational methods for pattern generation and particle dynamics simulation.

2 Methods

2.1 Overview

Compladni is a computational simulation of Chladni patterns from raw audio. The system is built on three main components: audio processing, physics simulation, and visualization. The audio processing component is responsible for converting the raw audio into a sequence of frequencies and amplitudes. The physics simulation component takes these frequencies and amplitudes as input, solves the states of the membrane at different positions, and simulates the motion of particles on the membrane. The visualization component renders the Chladni patterns as complete videos from the simulation results. The following figure shows an overview of the system.

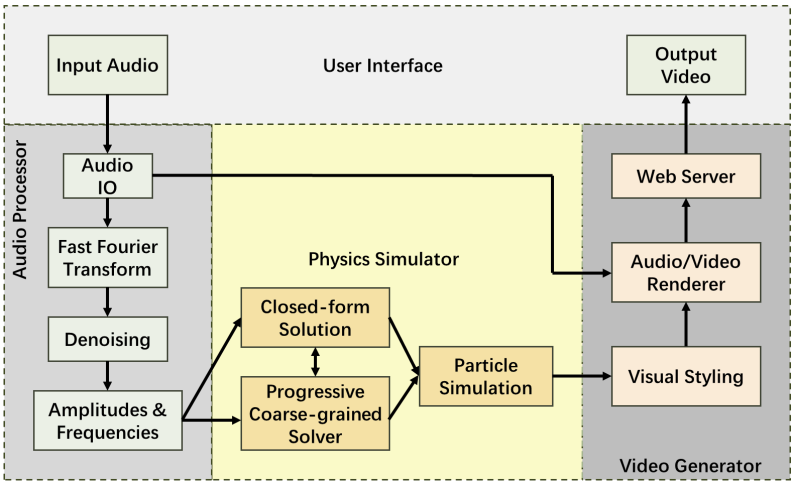


Fig 2.1.a: Overview of the system

2.2 Audio Processing

The audio processing module was developed to extract essential acoustic characteristics from audio files for Chladni pattern generation. The implementation utilized the `go-audio/wav` package for handling audio files, the `facebookresearch/demucs` package for music source separation, and the `go-dsp/fft` package for frequency analysis.

The processing pipeline consisted of several key steps:

2.2.1 Audio Input Processing

For audio input processing, the `go-audio/wav` package is utilized for handling the input audio files.

- Support was implemented for both mono and stereo WAV files.
- Channel separation was developed to enable independent analysis of stereo tracks.
- Systems for sample rate detection and buffer management were established to ensure smooth processing.

2.2.2 Music Source Separation

The music source separation stage used the `facebookresearch/demucs` package to decompose music into four stems: bass, drums, others, and vocals.

- The bass stem captured low-frequency instruments and bass lines.
- The drums stem isolated percussive and rhythmic components.
- The others stem represented melodic instruments and harmonics.
- The vocals stem separated human voice elements.

The detailed information is shown in 2.4.2.

2.2.3 Signal Analysis

2.2.3.1 Fast Fourier Transform (FFT)

For signal analysis, the **Fast Fourier Transform (FFT)** algorithm, provided by the `go-dsp/fft` package, was applied.

FFT is the fundamental algorithm we use for audio signal analysis, which can efficiently convert time-domain signals into their frequency components, allowing us to analyze "the spectrum or frequency content of a signal". [6]

Our system processes raw **PCM data** from WAV files, which initially exists as integer values ranging from **-32768** to **32767**. To prepare this data for FFT analysis, we first convert it to floating-point values normalized within the $[-1, 1]$ range, ensuring consistent processing across various input sources.

We use FFT every 4096 samples combined with a Hann window with 75% overlap between consecutive segments. This configuration enables us to accurately detect both subtle and dramatic frequency changes in the music while maintaining computational efficiency. The high overlap ratio (75%) ensures we capture rapid variations in the audio signal, which is crucial for generating responsive Chladni patterns that accurately reflect the musical dynamics.

Prior to FFT execution, each window undergoes two crucial preprocessing steps. First, we apply a **Hann Window** function to minimize spectral leakage, a common issue in frequency analysis. Second, we implement an adaptive **Noise Reduction** system based on the signal's **Root Mean Square (RMS)** value, effectively reducing background noise while preserving signal integrity. The `fft.FFTReal` function then processes this prepared data, generating a complex spectrum with 2048 frequency bins.

The resulting FFT output integrates seamlessly with downstream processing tasks. The complex spectrum is converted to amplitude information, enabling accurate peak detection for dominant frequency identification and RMS calculations for loudness measurement. These processed results directly feed into the Chladni pattern simulation system, providing the fundamental data required for pattern generation. This comprehensive approach ensures high-quality frequency analysis while maintaining computational efficiency, making it particularly suitable for real-time audio visualization applications.

2.2.3.2 Hann Window

To minimize spectral leakage and improve frequency analysis accuracy, **Hann Windowing** [7] with a 75% overlap was implemented. The Hann window is a cosine-based function expressed mathematically as:

$$w(n) = 0.5 \cdot \left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right),$$

where n is the sample index and N is the window length. This windowing approach significantly reduces spectral leakage and ensures smooth temporal transitions.

2.2.3.3 Noise Reduction

The noise reduction system implements a sophisticated adaptive **noise gate with soft-knee** characteristics, designed to effectively minimize background noise while preserving the integrity of the audio signal. Our implementation processes the normalized floating-point audio data, operating on windows of 4096 samples to maintain consistency with the FFT analysis pipeline.

The system employs a dynamic thresholding approach based on the signal's overall characteristics. The threshold is calculated as 5% of the signal's RMS value, which provides an adaptive reference level that automatically adjusts to different input signals. This approach proves more robust than fixed-threshold systems, particularly when processing audio sources with varying dynamic ranges and noise floors.

The noise reduction process incorporates three distinct operational zones:

- In the sub-threshold zone (signal magnitude < threshold), the signal is fully attenuated to eliminate background noise.
- The super-threshold zone (signal magnitude > 2.0 × threshold) maintains the original signal completely, ensuring that strong, intentional signals remain unmodified.
- The intermediate zone (threshold < signal magnitude < 2.0 × threshold) implements a **soft-knee transition** using linear interpolation, calculated as:

$$output = input * \frac{ratio - 1.0}{1.0} \quad (ratio = \frac{|input|}{threshold})$$

This soft-knee implementation was chosen specifically to avoid the artificial-sounding abrupt transitions often associated with traditional noise gates. The gradual transition provides natural-sounding noise reduction while minimizing artifacts and maintaining musical dynamics, making it particularly suitable for musical applications where preservation of artistic content is crucial.

The effectiveness of our noise reduction system is further enhanced by its integration with the FFT analysis pipeline. By applying noise reduction before frequency analysis, we improve the accuracy of **Dominant Frequency Detection** and reduce computational overhead in subsequent processing stages. This integration also ensures that the noise reduction process adapts to both temporal and spectral characteristics of the input signal.

2.2.3.4 Root Mean Square (RMS)

The loudness calculation system employs the **Root Mean Square (RMS)**[8] method, providing a robust and perceptually relevant measure of audio signal intensity. This approach was chosen over peak detection or average amplitude methods because RMS better correlates with human perception of sound intensity and provides a more stable metric for continuous audio analysis.

RMS effectively represents the energy level of the signal and provides a good approximation of perceived loudness. The mathematical foundation of RMS calculation is expressed as:

$$\mathrm{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2},$$

where n is the number of samples in the analysis window and x_i represents the amplitude of each sample. This method provided a reliable measure of the signal's loudness.

Our implementation processes the audio data in synchronized windows of 4096 samples, matching the FFT analysis framework. This synchronization ensures computational efficiency and maintains temporal consistency across different analysis components. The system operates on normalized floating-point data (range $[-1, 1]$), which provides consistent results regardless of the input audio's bit depth or encoding format.

2.2.4 Dominant Frequency Detection

The dominant frequency detection system implements a sophisticated peak analysis algorithm that combines spectral magnitude analysis with temporal consistency constraints. This approach was developed to address the common challenges in musical frequency analysis, particularly the presence of harmonics and rapid frequency transitions in complex musical compositions.

Our system employs a **top-5 peak analysis** methodology, which proves more robust than single-peak detection for musical applications. The analysis operates on the magnitude spectrum obtained from the FFT output, with frequency resolution determined by:

$$frequencyResolution = \frac{sampleRate}{fftLength},$$

where $sampleRate$ represents the number of audio samples per second (Hz) and $fftLength$ represents the number of samples in each FFT window.

Then, the peak detection algorithm operates in several stages:

1. Peak Identification:

- Process the magnitude spectrum up to the Nyquist frequency ($\frac{fftLength}{2}$).
- Store peaks as (magnitude, frequency index) pairs.
- Implement a sliding window comparison to identify local maxima.

2. Top-5 Peak Selection.

- Select the 5 highest magnitude peaks.

3. Temporal Consistency Analysis:

- Track the previous frame's dominant frequency.
- Among the top 5 peaks, select the candidate that:
  a. Has magnitude ≥ 50% of the maximum peak
  b. Is closest to the previous dominant frequency
- This approach prevents frequency jumping while maintaining responsiveness

The five-peak tracking system provides three crucial advantages over single-peak detection: it manages harmonic content by distinguishing fundamental frequencies from overtones, enables smooth handling of note transitions by tracking multiple frequency components simultaneously, and enhances noise resilience through redundant peak information and temporal consistency checking. This comprehensive approach makes the system particularly effective for analyzing complex musical signals where multiple frequencies coexist and change dynamically.

2.2.5 Data Export (Optional)

Finally, optional data export functionality was introduced to facilitate downstream testing. Processed data could be exported in CSV format, with separate output files created for the left and right channels. The data structures were organized for seamless integration into physics simulations for Chladni pattern generation.

2.3 Physics Simulation

2.3.1 Overview

As we discussed in the background section, the standing wave on the membrane results in different vibration strengths at different positions. Parts of the membrane vibrate more vigorously while others vibrate less. The spatial distribution of the vibration strength is determined by the frequency of the excitation. If we randomly scatter a vast number of particles on the membrane and excite the membrane with a certain frequency, we can observe all particles moving towards some very specific areas, forming visually appealing symmetric and complex patterns. Here is a figure showing the initial random distribution of particles on the membrane and the final distribution after the membrane is excited by a frequency of 2000 Hz.

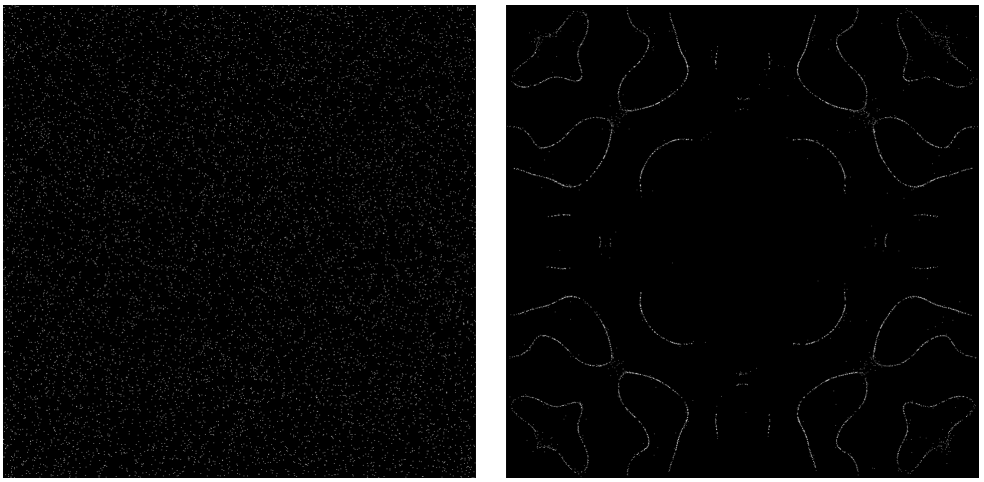


Fig 2.3.1.a: Initial state and final state

The initial state is a random distribution of particles on the membrane, and the final state is the Chladni pattern formed by the particles after the membrane is excited by a frequency of 2000 Hz.

In the physics simulation, we first randomly scatter particles on the membrane. Then, we repeatedly apply forces to the particles based on the vibration strength at their positions. In this process, the particles in positions with higher vibration strength will be pushed out of areas, while the particles in areas with lower vibration strength tend to stay still. As a result, after a sufficient number of iterations, the particles will all settle into stable positions, forming the Chladni pattern. The following figure shows a typical process of particle movement on an excited membrane with nodal lines represented in white.

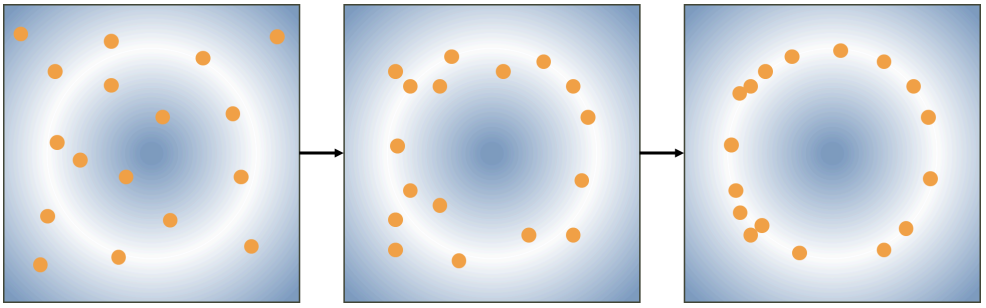


Fig 2.3.1.b: Typical process of particle movement on an excited membrane

The force applied to each particle is calculated based on the vibration strength at its position. In our implementation, we use a closed-form solution to calculate the vibration strength at each position, and generates a random force with varying direction, using the vibration strength as the magnitude. This random walk process is derived from the physical description of the particle behavior in physics research [9].

During the benchmarking of our algorithm, we found that the closed-form solution is computationally expensive, and is the performance bottleneck. Therefore, we developed a progressive coarse-grained solver that dramatically improves the performance of the physics simulation. In the following section, we will first introduce the closed-form solution, and then describe the details of the progressive coarse-grained solver.

2.3.2 Closed-form Solution

The closed-form solution is a mathematical solution to the vibration strength at each position on a membrane with specific shape and boundary conditions. In our implementation, we use a square membrane that is excited by a point source at its center. The closed-form solution is described in detail in modern physics textbooks [10]. The mathematical details behind the closed-form solution are not the focus of this paper, so we will not repeat the detailed derivation here. We implemented

the closed-form solution in an approximate manner so that it is programmable and within project scope. For simplicity, we will show the formal definition of our derived programmatic solution here.

2.3.2.1 Definition of function $\psi(p_x, p_y)$

The function $\psi(p_x, p_y)$ computes the vibration strength at a specific position (p_x, p_y) on a square plate by summing contributions from various wave modes. The vibration strength is influenced by the input frequency and accounts for various predefined physical properties of the system.

$$\psi(p_x, p_y) = \left(\frac{2}{L}\right)^2 \sum_{\substack{m,n=0 \\ m,n \text{ even} \\ (m,n)\neq(0,0)}}^{M} \frac{\cos\left(\frac{m\pi p_x}{P}\right)\cos\left(\frac{n\pi p_y}{P}\right)}{\sqrt{\left(\sqrt{\frac{f}{D}} - k_{mn}\right)^2 + \left(\frac{2\gamma\sqrt{\frac{f}{D}}}{L}\right)^2}}$$

- Particle Position (p_x, p_y): Represents the position of the particle on the plate in pixels.
- Input Frequency (f): Represents the input frequency of the simulation, in Hertz.
- Plate Size (L): Represents the length of one side of the square plate in meters.
- Half Plate Size in Pixels (P): Denotes half the plate size measured in pixels, used to scale the position coordinates p_x and p_y.
- Dispersion Constant (D): Represents a constant physical property of the material of the plate, describing the speed of the wave propagation.
- Energy Dissipation Constant (γ): Represents the damping factor that accounts for energy loss in the system, preventing the particles from oscillating indefinitely.
- Wave Number for Mode (m, n) (k_{mn}): Calculated for each mode (m, n), where m and n are even integers representing the number of half-wavelengths along the x- and y-axes, respectively.

$$k_{mn} = \frac{\pi}{L}\sqrt{m^2 + n^2}$$

- Maximum Mode (M): Represents the maximum mode number to be included in the summation.

2.3.2.2 Determining the movement of particles

As described in the overview section, the movement of particles is generated by a random walk process that is weighted by the vibration strength at their positions. In an actual audio to video simulation, we will also apply the amplitude of the input audio to the vibration strength, so that the movement of particles responses to the loudness of the input audio.

Given the vibration strength at each position, we calculate the force applied to each particle at each iteration by generating a random direction, and normalize it to the same scale as the vibration strength times audio amplitude.

However, most of music is not pure tones, and the frequency and amplitude of the input audio may change significantly over time. This could result in a situation where we can never form a stable Chladni pattern because of the varying frequency and insufficient number of iterations to converge. Therefore, we added a `StepScale` and `NumIterations` hyperparameter to create a layer of control over the random walk process, making the particle converges faster to the nodal lines, even when the input audio is noisy and highly unstable.

For clarity, we will use pseudocode to describe the process of determining the movement of particles. The input to the physics simulation is a list of particles, a list of frequencies, and a list of amplitudes. Extra parameters include `StepScale` and `NumIterations`. Note the difference between `NumIterations` and the actual number of simulation steps that we will perform. The actual number of simulation steps is equal to the length of the input frequency list. The `NumIterations` parameter is used to control the number of iterations that we will perform for each particle within one simulation step. Each iteration involves moving the particle once, based on the `StepScale` parameter. Here is the pseudocode:

```
def chladni_simulation(
    particles_count: int,
    frequencies: list[float],
    amplitudes: list[float],
    step_scale: float,
    num_iter: int
) -> list[list[Particle]]:

    # Initialize particles
    particles = initialize_particles(particles_count)
    time_series = [particles]

    # Simulate the Chladni pattern
    for frequency, amplitude in zip(frequencies, amplitudes):
        new_particles = step_simulation(particles[-1],
                                        frequency, amplitude,
                                        step_scale, num_iter)
        time_series.append(new_particles)
    return time_series

def step_simulation(
    particles: list[Particle],
    frequency: float,
    amplitude: float,
    step_scale: float,
    num_iterations: int
) -> list[Particle]:
    for particle in particles:
        for _ in range(num_iterations):
            vibration: float = psi(particle, frequency)
            unit: tuple[float, float] = random_direction()
            force = step_scale * amplitude * vibration * unit
            particle.move(force)
    return particles
```

2.3.2.3 Time complexity

For this unoptimized implementation, we iterate through all particles and calculate the force applied to each particle at each iteration. Let N be the number of particles, and I be the number of iterations. The total number of simulation units of this implementation is $N \times I$. However, a unit of simulation includes the calculation of the closed-form solution. Recall that we need to iterate through all possible combinations of (m, n) modes up to M, so we need to calculate the impact of M^2 mode combinations for each particle. Thus, the total number of simulation units is $M^2 \times N \times I$.

During our benchmarking, we found that our algorithm spent almost all of its time calculating the closed-form solution, and the random walk process only took a negligible amount of time. In preliminary testing, we also found that we need to have large enough M to get visually appealing patterns. For small M, the pattern of nodal lines changes less with different frequencies, and the pattern is less interesting. The figure below shows the typical pattern formed with different M values.

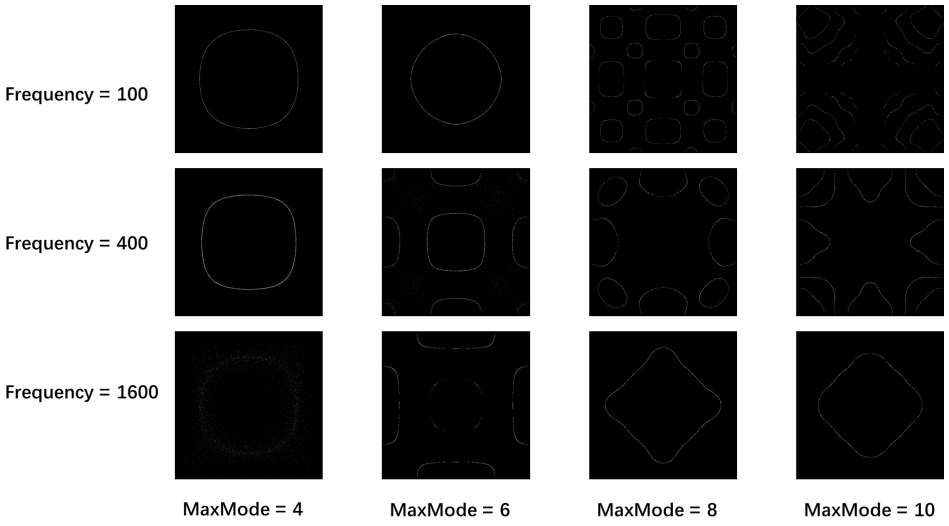


Fig 2.3.2.3.a: Typical pattern formed with different M values

To optimize the performance of the algorithm, we parallelized the simulation process. The parallelization brings significant performance improvement because the updates of particle positions are independent of each other. In our testing, the parallelized version of the algorithm can achieve a 10x speedup compared to the sequential version on a 20-core machine. However, with our innovative progressive coarse-grained solver, the seemingly significant speedup is not as pronounced.

2.3.3 Progressive Coarse-grained Solver

2.3.3.1 Coarse-grained solver

Since the goal of our project is to generate video demonstrations of the behavior of particles on a membrane, the precision of the simulation would become unnecessary if it goes beyond the resolution of the video. Therefore, a intuitive idea is to build a coarse grid matching the resolution of the video, and only calculate the vibration strength at the grid points.

During simulation, we can firtly test if the ψ value at the nearest grid point for each particle has been calculated. If so, we can directly use the value without recalculating it; if not, we calculate the ψ value using the closed-form solution and store it in the grid. Here is the pseudocode for the coarse-grained solver:

```
GRID = [[0, 0, 0, ...],
        [0, 0, 0, ...],
        ...,
        [0, 0, 0, ...]]

def coarse_grain_psi(x: float,
                     y: float) -> float:
    # Which grid does the particle fall in?
    grid_x: int = nearest_grid_coord(x)
    grid_y: int = nearest_grid_coord(y)
    # Already calculated? Just get the stored value!
```

```
    if is_calculated(GRID[grid_x][grid_y]):
        return GRID[grid_x][grid_y]
    # Not calculated? Calculate and store the value!
    else:
        vibration_strength = psi(grid_x, grid_y)
        GRID[grid_x][grid_y] = vibration_strength
        return vibration_strength
```

In this way, instead of doing $N \times I$ units of simulation, we instead need to do up to R^2 for each frequency, where R is the resolution of the video.

However, in actual audio files, the frequency of the audio signal changes over time, so we need to create a new coarse grid for each distinct frequency. To better reuse the coarse grids which contain previously calculated ψ values, we chose to also approximate the frequency of the audio signal by rounding it to the nearest multiple of Δf, where Δf is the frequency resolution of the coarse grid. In this way, we can reuse the coarse grids across different audio frequencies. Below is a figure showing the overall architecture of our coarse-grained solver.

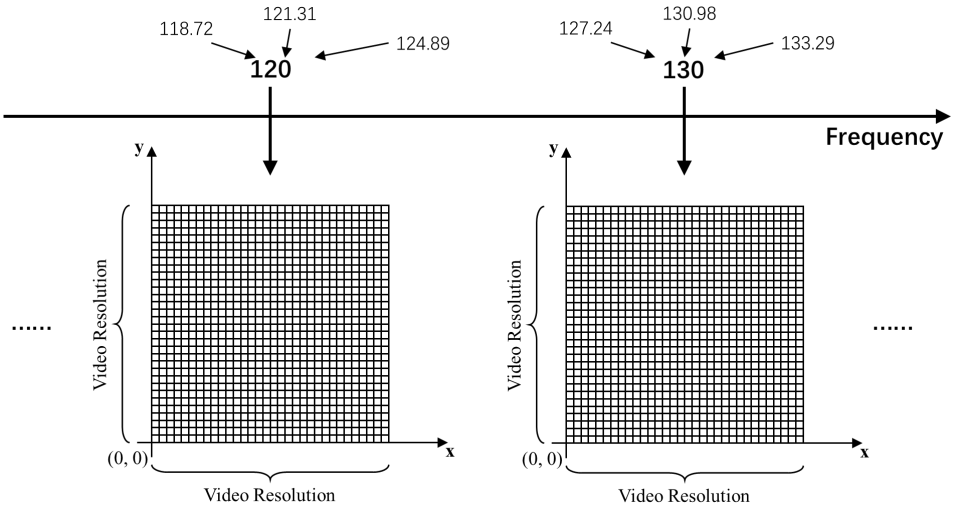


Fig 2.3.3.1.a: Overall architecture of the coarse-grained solver

This coarse-grained solver could present an improvement over the original implementation when $N \times I$ is large and R^2 is small for each corresponding frequency. However, in practice, audio files contain a wide range of frequencies, resulting the I value being relatively small for most frequencies, unless we round the frequency to ridiculously large Δf. Thus, the performance improvement is not as significant as expected.

2.3.3.2 Motivation of the progressive coarse-grained solver

Our invention of the progressive coarse-grained solver comes from the observation that particles in the Chladni pattern tend to settle and accumulate in very specific areas, the nodal lines. The vibration strength near these nodal lines are low and need to be calculated more precisely to form the correct pattern. However, for areas far away from the nodal lines, the vibration strength is high and the particle will never accumulate there. The precision of the ψ values in far away areas is not as important because most particles will not be in these areas after several iterations of random walk.

This observation motivates us to only calculate precise ψ values for points near the nodal lines, and use less precise ψ values for areas far away from the nodal lines. This will only negligibly compromise the quality of the Chladni pattern, because very tiny proportion of particles will be in these far away areas after several iterations of random walk.

Based on the motivation described above, we propose a progressive coarse-grained solver that progressively adjusts the precision of the ψ values based on the density of particles in different areas.

2.3.3.3 Implementation of the progressive coarse-grained solver

Instead of building a $R \times R$ coarse grid initially, where R is the resolution of the video, we start with a very low resolution 2×2 grid holding the ψ values of at each grid point, along with a counter at each grid point recording the access to that grid point. For every incoming particle, we first approximate its position to match a grid point. If the ψ value at the grid point has been calculated, we use the value directly and increment the counter at the grid point; if not, we calculate the ψ value using the closed-form solution and store it in the grid.

Then, we predefine a hyperparameter T that indicates a threshold of the counter value for a grid point to be considered as a high density area. If the counter value at a grid point is greater than T, we consider the area as a high density area and believe that a nodal line is present in the area. To refine the ψ value at this high density grid point, we then build a smaller 2×2 grid inside the high density grid point. If we call the intial 2×2 grid the level 0 refinement, we can call the new 2×2 grid the level 1 refinement, providing a doubled precision. Thus, for any future particle that falls into this area, instead of getting the ψ value from the larger 2×2 grid, we will go into the smaller 2×2 grid and do a more precise calculation.

The above process cound be repeated multiple times, and results in a higher and higher level of refinement in certain areas where nodal lines are present. The process is repeated until the resolution of the finest grid reaches the resolution of the video. Here is a visual example of this progressive coarse-grained solver in action.

1. Assume that the resolution of the video is 32×32 pixels and the threshold T is 10. We start with a 2×2 grid, and the counter at each grid point is initialized to 0.

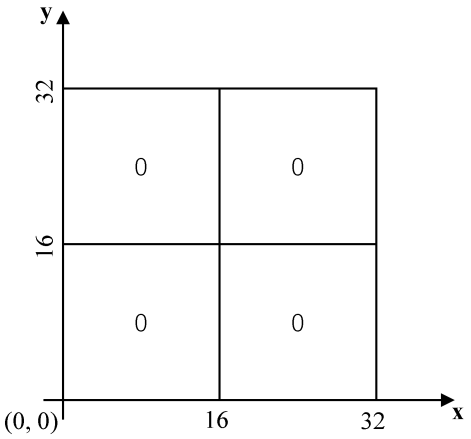


Fig 2.3.3.1.b: Initial state of the coarse-grained solver

2. A particle with coordinates $x = 22, y = 13$ is scattered onto the membrane and is placed into the bottom right grid point, which has not been calculated before. We calculate the ψ value at this grid point, store the value in the grid, and increment the counter by 1.

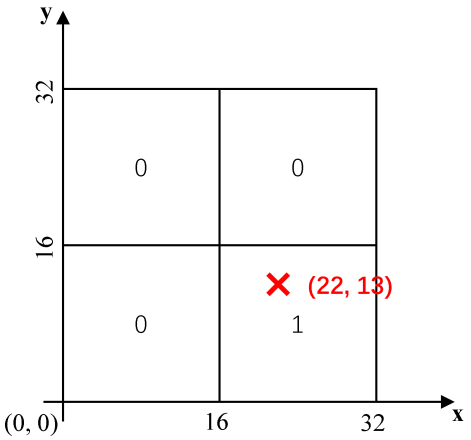


Fig 2.3.3.1.c: First iteration of the coarse-grained solver

3. The above process is repeated for a few more iterations. Since the resolution is low, most of the particles will get a previously calculated ψ value. After several iterations, we can see that the counter at the bottom right grid point is meets the threshold $T = 10$.

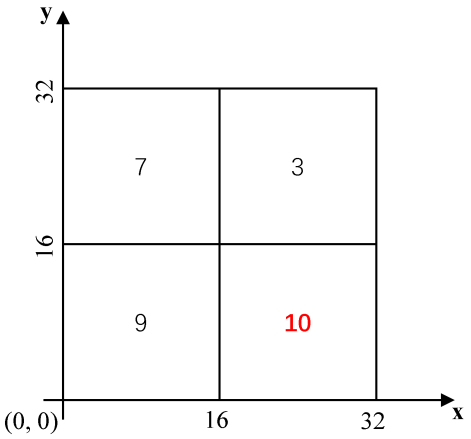


Fig 2.3.3.1.d: After several iterations

4. We then build a smaller 2×2 grid inside the high density grid point and initialize the counter at each new grid point to 0.

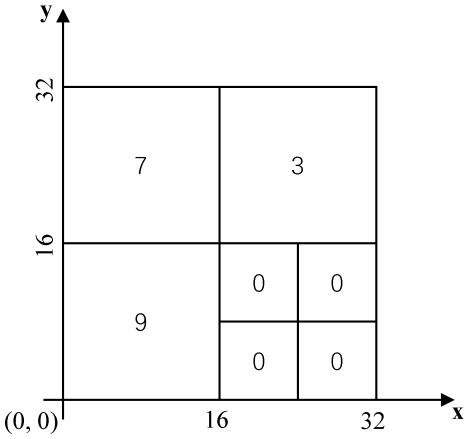


Fig 2.3.3.1.e: Build a smaller grid inside the high density grid point

5. A particle with coordinates $x = 20, y = 14$ is scattered onto the membrane and is placed into the smaller grid point which we just created and has not been calculated before. We calculate the ψ value at this grid point, store the value in the smaller grid, and increment the corresponding counter by 1. Since the resolution of the smaller grid is higher, the ψ value we get is more precise.

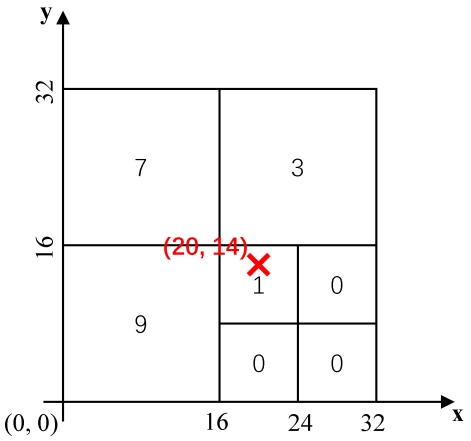


Fig 2.3.3.1.f: A particle is scattered onto the membrane

6. The above process is repeated for lots of iterations, and particles will become more and more concentrated in at nodal lines. In the following figure, we drew an example nodal line in red to indicate the area where particles are more likely to accumulate. We can see that the resolution of the grid near the nodal lines is higher, so the ψ value is more precise.

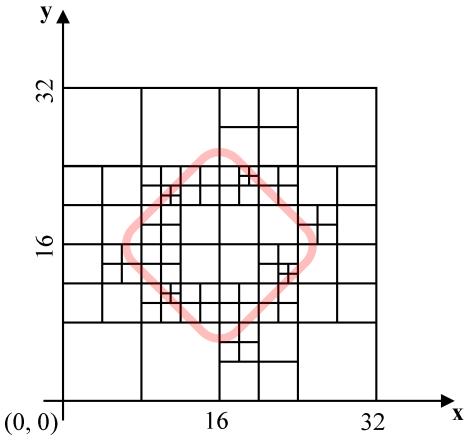


Fig 2.3.3.1.g: After several iterations

7. Finally, after enough iterations, we can see that the Chladni pattern is formed and almost all particles are in the nodal lines. The resolution of the grid near the nodal lines will grow larger and larger until it reaches the resolution of the video.

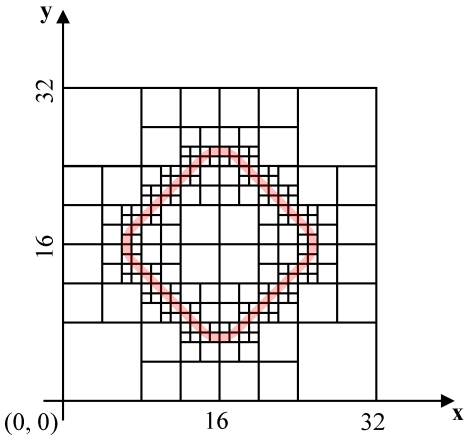


Fig 2.3.3.1.h: After enough iterations

Here is a pseudocode showing the implementation of the progressive coarse-grained solver:

```
class ProgressiveCoarseGrid:
    # Where psi values are stored
```

```
    values = [[None, None, None, ...],
              [None, None, None, ...],
              ...,
              [None, None, None, ...]]
    # How many times the grid point has been accessed
    counter = [[0, 0, 0, ...],
               [0, 0, 0, ...],
               ...,
               [0, 0, 0, ...]]
    # Children grids, used for refinement
    children = [[None, None, None, ...],
                [None, None, None, ...],
                ...,
                [None, None, None, ...]]

GRID = ProgressiveCoarseGrid() # One grid for one frequency

def pcg_psi(x: float, y: float, grid = GRID) -> float:
    # PCG: Progressive Coarse-Grained Solver

    # Find the grid point that the particle falls in
    g_x, g_y = nearest_grid_coord(grid, x, y)

    # Threshold reached? Do refinement
    if grid.counter[g_x][g_y] >= THRESHOLD:
        # Is the children grid initialized?
        if grid.children[g_x][g_y] is None:
            grid.children[g_x][g_y] = ProgressiveCoarseGrid()
        # Recursive call to the children grid
        return pcg_psi(x, y, grid.children[g_x][g_y])

    # Calculated? Get the stored value and increment the counter
    if grid.values[g_x][g_y] is not None:
        grid.counter[g_x][g_y] += 1
        return grid.values[g_x][g_y]

    # Not calculated? Calculate the value and store it in the grid
    else:
        vibration_strength = psi(g_x, g_y)
        grid.values[g_x][g_y] = vibration_strength
        grid.counter[g_x][g_y] = 1
        return vibration_strength
```

Similarly, since actual audio files contain a wide range of frequencies, we can round the frequency to the nearest multiple of Δf to reuse the coarse grids across different audio frequencies. Here is a figure showing the overall architecture of the progressive coarse-grained solver:

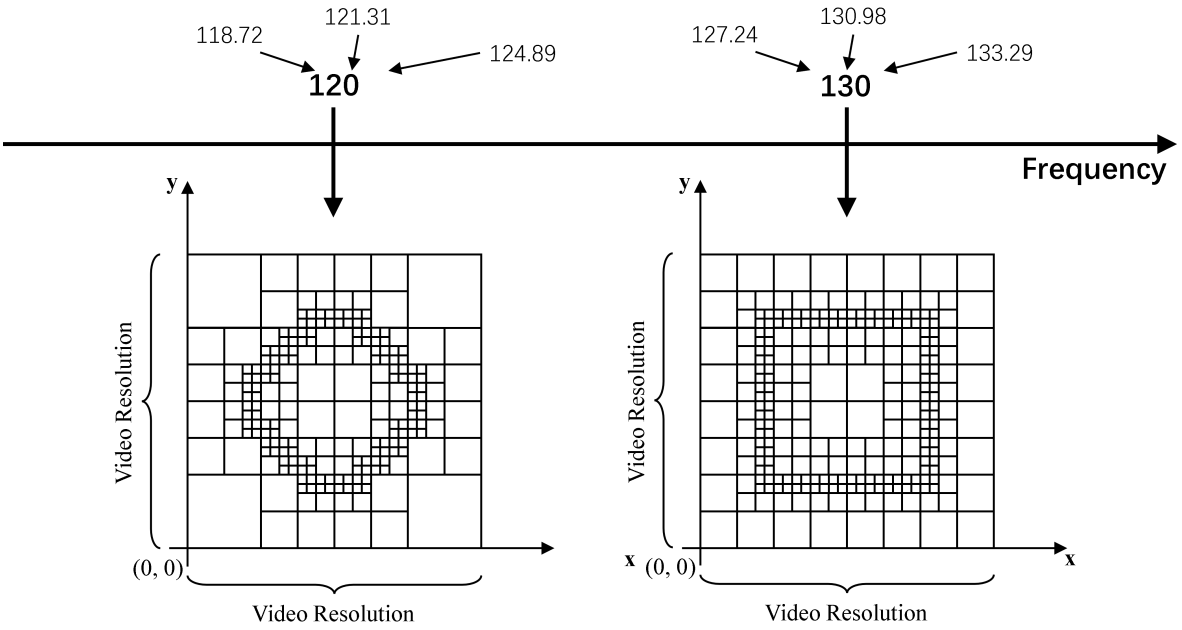


Fig 2.3.3.1.i: Overall architecture of the progressive coarse-grained solver

2.3.3.4 Estimating time complexity

In our previous implementation where we have an $R \times R$ coarse grid, the number of times we need to calculate the ψ value is up to R^2. As it turns out, in our progressive coarse-grained solver, the number of times we need to calculate the ψ value is proportional to the number of particles. Here is how we derive this conclusion.

Say that we are about to simulate the Chladni pattern of a certain frequency with the video resolution R. The length of nodal lines formed by this frequency is L pixels. Recall that particles will accumulate in the nodal lines, so the final situation is that all grids that touches the nodal lines will be refined to the resolution of the video. On average, every two pixel in the length of nodal lines will correspond to one maximum refinement level 2×2 grid. The results in a total number of $2L$ calculations of ψ values for the maximum refinement level. Similarly, on average, every two 2×2 grids in the nth refinement level will correspond to one 2×2 grid in the $(n-1)$th refinement level. Thus, the total number of calculations of ψ values on level n is half of that on level $n+1$.

For a video of resolution R, the maximum number of refinement levels is $\log_2 R$. Therefore, the total number of calculations of ψ values can be calculated as:

$$\sum_{i=0}^{\log_2 R} 2L\left(\frac{1}{2}\right)^i < 4L$$

Since L is the length of nodal lines in pixels, it is proportional to the resolution of the video R. Therefore, the total number of calculations of ψ values can be written as $4 \times \text{constant} \times R$. Let's assume that the nodal line forms a perfect square with side length $\frac{R}{2}$, then $L = 4 \times \frac{R}{2} = 2R$. Therefore, the total number of calculations of ψ values is $8R$, which is linear to the resolution of the video.

In this way, we reduce the total number of calculations of ψ values from R^2 to $8R$. For a typical video of resolution 1024×1024, the total number of calculations of ψ values is reduced from over a million to less than 10 thousand, which is a 100x speedup. This trend is very obvious if we look at this figure:

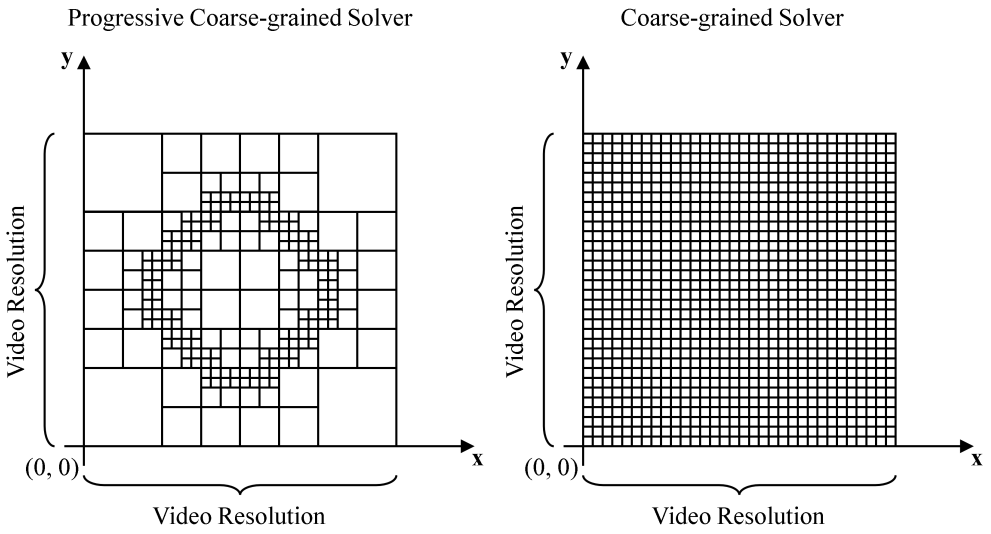


Fig 2.3.3.4.a: Grid visualization comparison

This theoretical trend matches the performance improvement we observed in our benchmarking. In the physics simulation with 10000 particles, 10000 iterations, 1024×1024 resolution at a fixed frequency, the serial version of the progressive coarse-grained solver is even faster than the parallel version of the original implementation. Here is a figure showing the comparison of the performance of the two implementations.

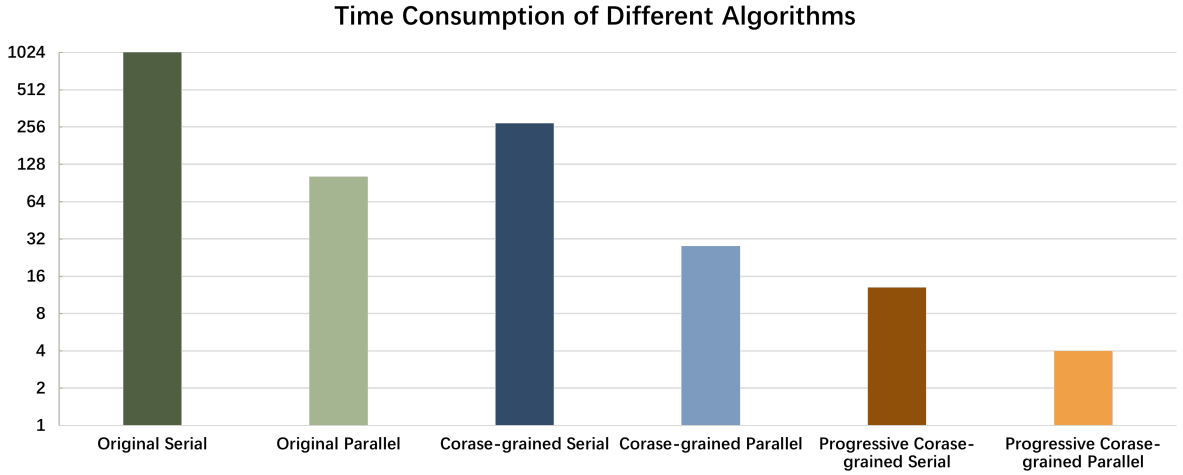
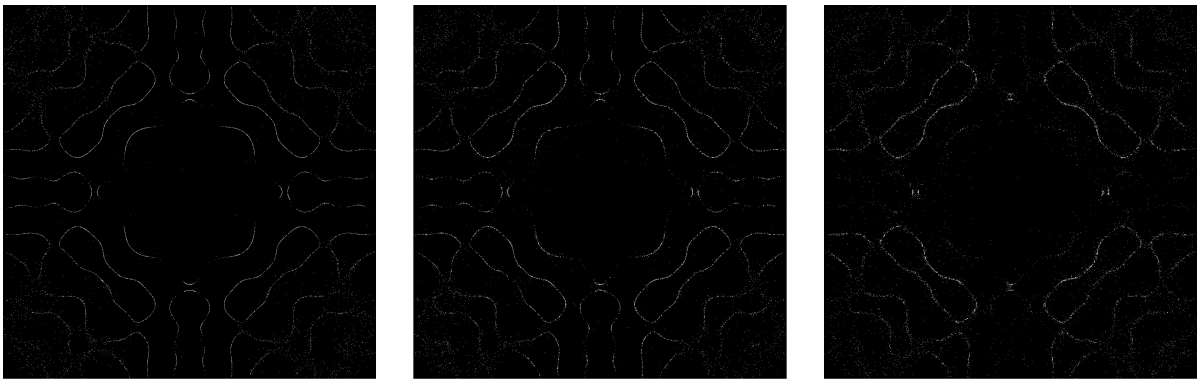


Fig 2.3.3.4.b: Comparison of the performance

Y-axis: Time taken in seconds in logarithmic scale

2.3.3.5 Quality Comparison

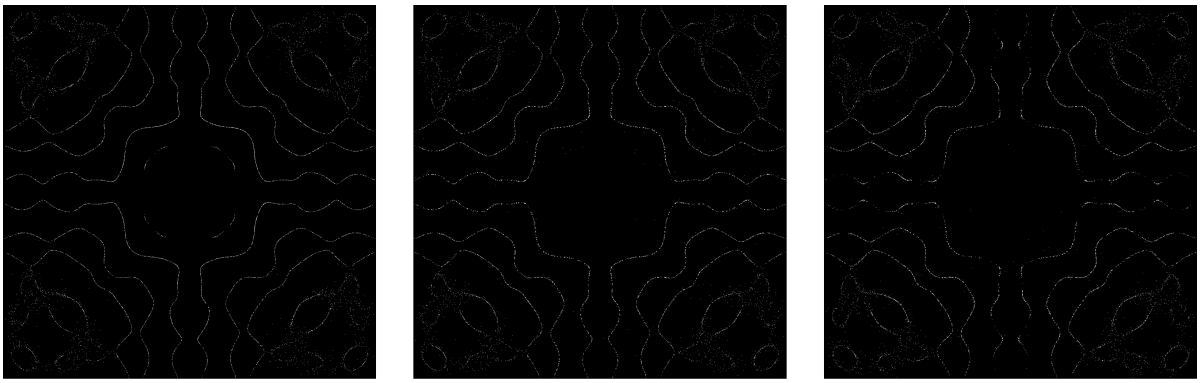
It turns out that the quality of the Chladni pattern drops when we use more efficient solvers. The original implementation is the most accurate, forming the most complete Chladni pattern with sharpest nodal lines. The coarse-grained solver is the second most accurate, but it may miss some delicate features and may make certain nodal lines less clear. The accuracy of the progressive coarse-grained solver varies depending on the number of refinement levels it has, but it is generally the least accurate. However, if we simulate on larger scale, the quality of the progressive coarse-grained solver will gradually improve and be comparable to the coarse-grained solver.



Precise Simulation | Coarse-Grained | Progressive Coarse-Grained

Fig 2.3.3.5.a: Quality comparison when the number of iterations is small

Simulated on 100 iterations. We can see that the coarse-grained solver is more blurry than the precise implementation. The progressive coarse-grained solver is most blurry, and is missing some nodal lines.



Precise Simulation | Coarse-Grained | Progressive Coarse-Grained

Fig 2.3.3.5.b: Quality comparison when the number of iterations is large

Simulated on 1000 iterations. We can see that the precise implementation now forms some very fine features, which is not visible in the coarse-grained solver. The overall quality of the progressive coarse-grained solver is now comparable to the coarse-grained solver.

2.3.3.6 Parallelization of the progressive coarse-grained solver

Since the progressive coarse-grained solver uses a 2D grid to store the ψ values, we will need careful handling of the grid to ensure that the parallelization is safe and efficient, so that each cell of the grid can be accessed by multiple threads without causing race conditions, repeated calculations, or other issues.

To understand this, let's first recap all operations we need to perform on the grid in pseudocode:

```
class ProgressiveCoarseGrid:
    values: list[list[float]]
    counter: list[list[int]]
    children: list[list[ProgressiveCoarseGrid]]

    def get_psi(self, x: int, y: int) -> float:
        if self.children[x][y] is not None:
            return self.children[x][y].get_psi(x, y)
        else:
            if not reached_max_refinement(self.counter[x][y]):
                self.counter[x][y] += 1
            return self.values[x][y]

    def set_psi(self, x: int, y: int):
```

```
        self.values[x][y] = psi(transform(x), transform(y))
        self.counter[x][y] = 1

    def create_child(self, x: int, y: int):
        self.children[x][y] = get_child(*args)
```

As we can see, all operations on the grid involves modifying the cell of the grid or its children. To make the parallelization safe, we need to ensure that no two threads modify the same grid cell at the same time. We can achieve this by using a lock on each grid cell.

When a thread wants to access a grid, we first check if the grid cell is locked. If it is, we wait for the lock to be released. If it is not, we proceed by checking if the current access needs a lock. If it does, we lock the grid cell and proceed. If it does not, we proceed without locking.

An access may need a lock if it reads from an underpopulated grid cell (and increments the counter), or if it writes to an uncalculated grid cell, or if it creates a child grid. An access does not need a lock if it reads from a populated grid cell with a valid child grid, or if it reads from a grid cell that has already reached the maximum refinement level.

To further optimize the parallelization, we defined a loose threshold to disable locking when reading from a very underpopulated grid cell. For example, if the predefined refinement threshold is 100 and the counter at a grid cell is only 7, we will not lock the grid cell when reading from it. This is because the grid cell is very unlikely to get to above the threshold during the reading and incrementing process. This optimization reduces the overhead of locking and unlocking the grid cell, and speeds up the simulation by 5% in preliminary testing.

2.4 Render and Visualization

2.4.1 Basic Rendering Method

In designing the visualization strategy for Chladni patterns, we faced two seemingly contradictory objectives: the need to precisely represent each particle's position to reflect authentic physical motion, while simultaneously creating a soft, ethereal visual effect that captures the flowing nature of sound waves. To resolve this challenge, we embraced a Dual-Layer Rendering approach as our core visual strategy.

This dual-layer structure draws inspiration from the natural phenomenon of light scattering. Much like observing the night sky, where each star possesses both a brilliant core and a gentle halo, our visualization endows each particle with two visual layers: a sharp, bright nucleus and a diffused glow layer. The bright nucleus employs linear attenuation to ensure positional accuracy, while the glow layer follows an exponential decay function:

$$GlowIntensity \propto e^{(-3d/R)}$$

where 'd' represents the distance from the particle's core, and 'R' defines the glow radius. This decay pattern not only creates natural light diffusion effects but also enables smooth transitions when particles cluster together.

We then transforms each frame into a celestial tableau, where particles dance like stars responding to the music's invisible forces. The rendering system processes these frames in parallel, orchestrating a symphony of visual elements:

```
go func(start, end int, ch chan []*image.RGBA) {
    batch := make([]*image.RGBA, end-start)
    for i := start; i < end; i++ {
        batch[i-start] = r.RenderSingleFrame(i, voicePart)
    }
    ch <- batch
}(startIdx, endIdx, channels[i])
```

After these settings, you can conclude from Fig 2.4.1 that not only does our dual-layer rendering approach effectively capture the precise particle positions, but it also creates a visually captivating representation of the wave dynamics. The bright nuclei of the particles clearly mark their exact locations while the surrounding glow layers blend smoothly to reveal the underlying wave patterns. The parallel processing implementation ensures this visual quality is maintained even at high frame rates, producing fluid animations that accurately represent the physical phenomena.

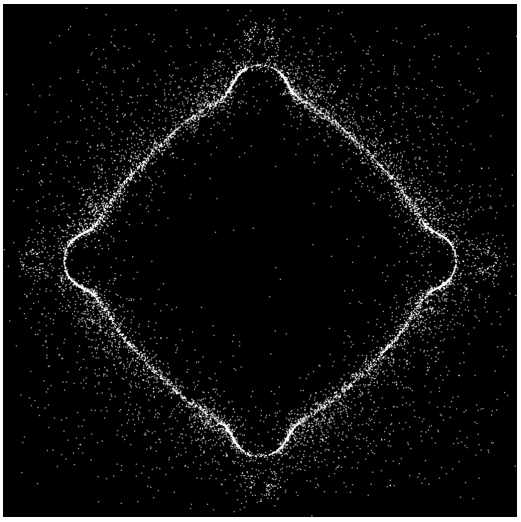


Fig 2.4.1: Rendering result demonstration

The white particles represent the final rendering result of our system, showing both the precise positions and the overall visual effect of the simulation

2.4.2 Voice Part Separation Analysis

For the audio separation task, we utilized Demucs v4, a pre-trained model that demonstrates state-of-the-art performance in music source separation. The model effectively separated our audio tracks into distinct components - vocals, drums, bass, and other accompaniments, achieving satisfactory results.[11][12]

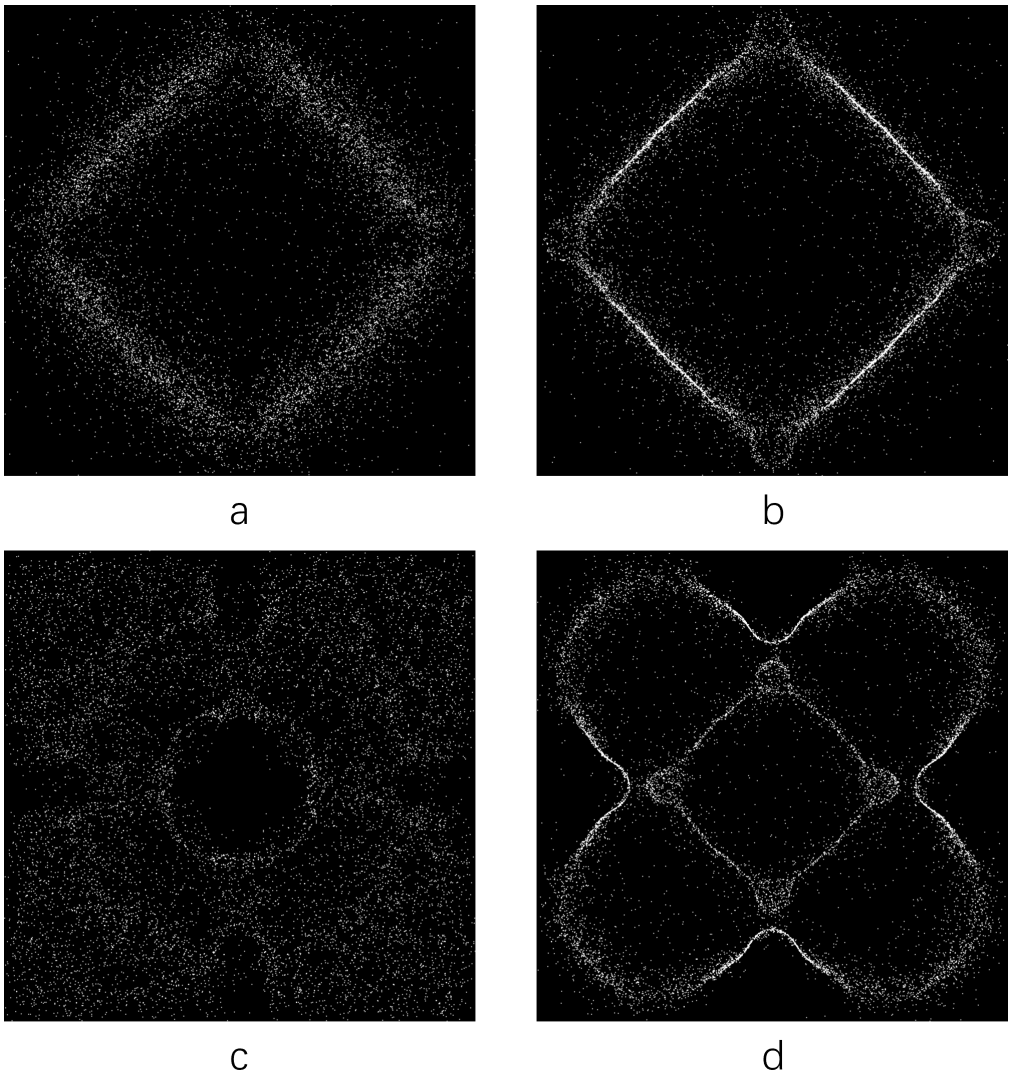


Fig 2.4.2.a: Visualization of different voice parts at the same timestamp

Multiple panels showing distinct particle distributions: (a) Bass track with concentrated patterns in lower frequencies; (b) Drum track displaying focused, rhythmic distributions; (c) Other instruments showing dispersed patterns across frequency ranges; (d) Vocal track demonstrating broad frequency range distributions with characteristic clustering

As shown in Fig. 2.4.2.a, the particle patterns of different voice parts at the same timestamp exhibit distinct characteristics. From left to right, the bass track (a) and drum track (b) show more concentrated particle distributions, which aligns with our expectation for low-frequency components. In contrast, the other instruments (c) and vocal track (d) display more dispersed patterns, reflecting their broader frequency ranges. These distinct visualizations at the identical moment demonstrate that each pattern responds exclusively to its corresponding audio component, validating the effectiveness of our voice separation approach.

But this is just a start. In crafting our visualization's color palette, we found inspiration from an interesting coincidence - our example track comes from Akina Nakamori's 1985 album "CRIMSON." This serendipitous connection led us to choose crimson (`{220, 20, 60, 255}`) as the color for vocal patterns, paying a subtle homage to the J-pop diva's iconic album. As shown in Fig 2.4.2b, this choice not only carries this meaningful reference but also proves to be visually striking.

Building around this central crimson theme, we carefully selected complementary colors for other voice parts to create a harmonious visual ensemble. The deep violet (`{75, 36, 115, 255}`) for bass patterns provides a rich, grounding presence, while the electric blue (`{54, 137, 160, 255}`) for drums adds dynamic energy to the visualization. The accompanying instruments are rendered in a

subtle vapor cyan (`{102, 187, 178, 255}`), which gracefully fills the spaces between the more prominent elements without overshadowing them.

Through our dual-layer rendering approach, these colors interact and blend naturally as particles move across the plate. Each voice part maintains its distinct identity while contributing to a cohesive whole - much like how the instruments in Nakamori's tracks weave together to create her signature sound. The rendering parameters for each voice part are carefully tuned to this color scheme, with the core and glow effects enhancing the natural interplay between different musical elements.

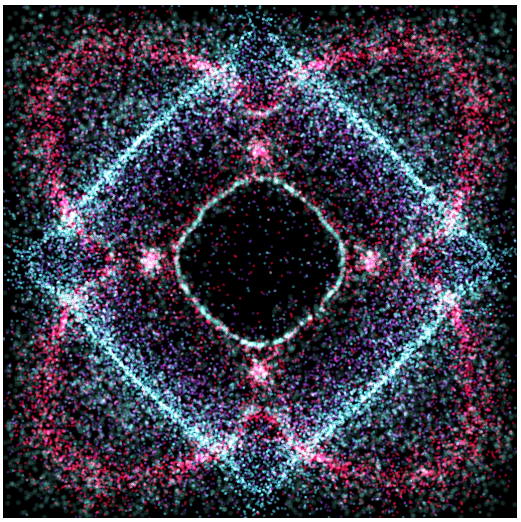


Fig 2.4.2.b: Color scheme visualization

Complete visualization demonstrating the interaction of multiple color layers: Crimson (#DC143CFF) for vocals, Deep violet (#4B2473FF) for bass, Electric blue (#3689A0FF) for drums, and Vapor cyan (#66BBB2FF) for other instruments, showing how different colors blend while maintaining distinct voice part identities

2.4.3 Rshiny Design

We developed an interactive Shiny application to demonstrate the Chladni pattern generation system. As shown in Fig 2.4.3, the graphical user interface adopts an intuitive layout, dividing the functionality into two main sections: a parameter control panel and a result display area.

The control panel on the left allows users to upload audio files in WAV format and adjust key parameters through a series of numeric input boxes, including the plate's physical dimensions, image resolution, particle count, and other physical simulation parameters. To optimize computational performance, we introduced a coarse simulation option, where users can balance calculation speed and accuracy through a multi-level resolution refinement strategy.

The main panel on the right, as demonstrated in the screenshot, employs a hierarchical video display layout. The complete Chladni pattern animation is shown at the top, while the visualizations for bass, drums, vocals, and other instruments are displayed separately below. A progress bar and status indicators allow users to monitor the processing progress in real-time. This layout design not only facilitates observation of the overall effect but also enables easy comparison of vibration characteristics across different voice parts, providing an intuitive analytical tool for music visualization research.

The system's responsive design ensures a good user experience across different devices. Through this demonstration interface, users can conveniently explore how different parameters influence the formation of Chladni patterns and gain a deeper understanding of the relationship between sound and physical vibration patterns.

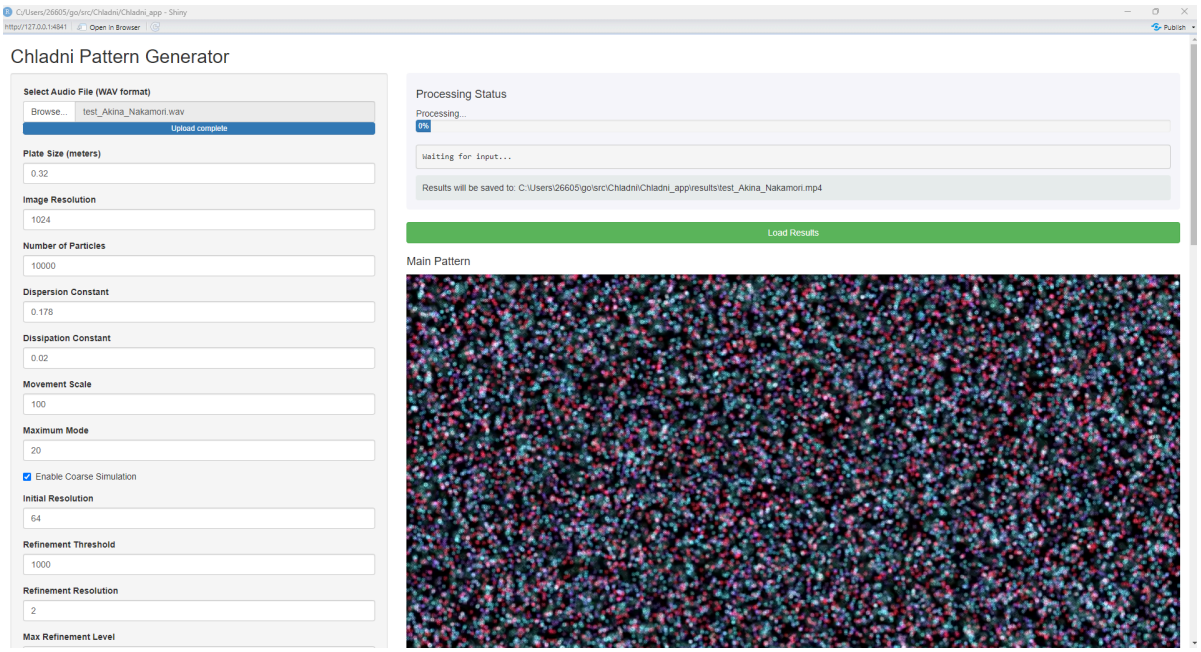


Fig 2.4.3: Interactive visualization demonstration using R Shiny

3 Team Contributions

Claude:

- Project idea and proof of concept
- Essay: Methods Overview, Physics Simulation
- Code: Physics Simulation Module, Preliminary Visualization, Integration, Architecture Planning, Optimization
- Presentation: Introduction, Methods Overview, Physics Simulation

Luci:

- Essay: Visualization
- Code: Visualization Module, RShiny, Integration, Code Optimization
- Presentation: Visualization

ShiYu:

- Essay: Introduction, Audio Processing
- Code: Audio Processing Module
- Presentation: Audio Processing

4 References

1. “Ernst Chladni.” Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc., Last edited on 30 November 2024. https://en.wikipedia.org/wiki/Ernst_Chladni ↩

2. R. C. Colwell, "Chladni figures on square plates," Journal of the Franklin Institute, vol. 221, no. 5, pp. 635-652, 1936. ↩

3. R. C. Colwell, A. W. Friend, and J. K. Stewart, "The Vibrations of Symmetrical Plates and Membranes," The Journal of the Acoustical Society of America, vol. 10, no. 1, pp. 68-73, 1938. ↩

4. "CYMATICS: Science Vs. Music - Nigel Stanford," YouTube. [Online]. Available at: https://www.youtube.com/watch?v=Q3oItpVa9fs ↩

5. "Musical Fire Table!" YouTube. [Online]. Available at: https://www.youtube.com/watch?v=2awbKQ2DLRE ↩

6. Heideman, M. T.; Johnson, D. H.; Burrus, C. S. Gauss and the history of the fast Fourier transform. IEEE ASSP Magazine. 1984, 1(4): 14–21. ↩

7. “Hann function.” Wikipedia. [Online]. Last edited on 16 April 2024. Available at: https://en.wikipedia.org/wiki/Hann_function ↩

8. “Root mean square.” Wikipedia. [Online]. Last edited on 19 November 2024. Available at: https://en.wikipedia.org/wiki/Root_mean_square ↩

9. Arango, Jaime, and Carlos Reyes. "Stochastic models for chladni figures." Proceedings of the Edinburgh Mathematical Society 59.2 (2016): 287-300. ↩

10. Serway, Raymond A., and John W. Jewett. Physics for scientists and engineers with modern physics. Vol. 2. Cengage Learning, 2019. https://phys.libretexts.org/Bookshelves/Waves_and_Acoustics/The_Physics_of_Waves_(Goergi)/11%3A_Two_and_Three_Dimensions/11.03%3A_Chladni_Plates ↩

11. Rouard, S., Massa, F., & Défossez, A. "Hybrid Transformers for Music Source Separation." In Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2023. ↩

12. Défossez, A. "Hybrid Spectrogram and Waveform Source Separation." In Proceedings of the International Society for Music Information Retrieval (ISMIR) Workshop on Music Source Separation, 2021. ↩